Tipos Algebraicos Recursivos: Polinomios

Taller de Álgebra I

Segundo cuatrimestre de 2015

Polinomios

En esta clase vamos a trabajar con polinomios. Para hacer las cosas fácil, trabajaremos en "$\mathbb{R}[x]$". Es decir, en `Float`$[x]$.

Hoy veremos lo necesario para entender y programar lo siguiente:

```
*Main> let p = 2 - 3*x + 4*x^4
*Main> p
2.0x^0 + -3.0x^1 + 4.0x^4
*Main> derivada p
-3.0x^0 + 16.0x^3
*Main> evaluar (derivada p) 10.0
15997.0
```

Hay muchas maneras de representar polinomios. Una de ellas es a partir de monomios ax^n, sumas y productos de polinomios. Es decir,

```
data Polinomio = Mono Float Integer
               | Suma Polinomio Polinomio
               | Producto Polinomio Polinomio
```

En rigor, con monomios y sumas alcanza, pero.. ¿por qué es necesario definir el producto? ¿qué queremos modelar?
Queremos poder diferenciar $(x^2 + 2)(x + 3)$ de $x^3 + 3x^2 + 2x + 5$?
Queremos poder modelar simplifaciones de este estilo? $x + x + 2x \longrightarrow 4x$

Evaluando polinomios

```
data Polinomio = Mono Float Integer
               | Suma Polinomio Polinomio
               | Producto Polinomio Polinomio
```

¿Cómo se escribe en Haskell lo siguiente?

- $3x^4$
- 7
- $2x + 7$
- $(x^2 + 2)(x + 3)$

A evaluar polinomios

Escribir `evaluar :: Polinomio -> Float -> Float`
que dado un polinomio p y un `Float` z devuelva $p(z)$.

coeficientes

Como toda la información de un polinomio está en los coeficientes, queremos poder calcular los coeficientes de un polinomio. Los coeficientes los podemos poner en una lista, empezando por el de grado 0, luego el de grado 1, etc. Es necesario mantener esta lista sin ceros a la derecha.

Por ejemplo

coeficientes de 7: `[7]`,
coeficientes de $(2x + 7 - 3x)(x^2)$: `[0.0,0.0,7.0,-1.0]`,
coeficientes de 0: `[]`

Programar

La función `coeficientes :: Polinomio -> [Float]`.

Para pensar el caso del producto, suponiendo que se desea multiplicar dos polinomios $\mathbf{p} \cdot \mathbf{q}$, utilizar la siguiente propiedad:
$p = (a_0 + a_1 x + \cdots + a_n x^n) = (a_0 + x \cdot \tilde{p})$ luego,
$\mathbf{p} \cdot \mathbf{q} = (a_0 + x \cdot \tilde{p}) \cdot q = a_0 \cdot q + x \cdot \tilde{p} \cdot q$
(notar que $\tilde{p} = a_1 + a_2 x + \cdots + a_n x^{n-1}$ es un polinomio)

Derivar!

Programar la función `derivada :: Polinomio -> Polinomio`

Num y Show

- La clase `Num` es la de los tipos numéricos. Más precisamente, son los tipos para los que se puede sumar, multiplicar, restar. En términos matemáticos, son los anillos.
- Observación: $x - y = x + (-y)$, así que para restar alcanza con saber sumar y tener opuestos.
- Así como la clase `Show` es la de los tipos que tienen definida la función `show`, la clase `Num` es la de los que tienen `(+)`, `(*)`, `negate`, `signum`, `abs`.
- La función $f : \mathbb{Z} \to A$ para un tipo A de la clase `Num` se llama `fromInteger`. Es decir, convierte un número a un Polinomio en este caso.

Con lo anterior, completar la los (...)

```
instance Num Polinomio where
    (+) p q = (...)
    (*) p q = (...)
    negate p = (...)
    fromInteger n = (...)
    abs p = undefined
    signum p = undefined
```

```
instance Show Polinomio where
    show p = (...) -- Utilizar la lista de coeficientes
```

Usando polinomios

```
*Main> let p = Suma (Mono 1 0) (Mono 2 1)
*Main> p
1.0x^0 + 2.0x^1
```

```
*Main> let q = (Mono 2 0) + (Mono (-3) 1) + (Mono 1 2)
*Main> q
2.0x^0 + -3.0x^1 + 1.0x^2
```

```
*Main> p + q
3.0x^0 + -1.0x^1 + 1.0x^2
*Main> p * q
2.0x^0 + 1.0x^1 + -5.0x^2 + 2.0x^3
```

```
*Main> q^3
8.0x^0 + -36.0x^1 + 66.0x^2 + -63.0x^3 + 33.0x^4 + -9.0x^5 + 1.0x^6
```

```
*Main> let x = Mono 1 1
*Main> let p = 1 + x - 3 * x^2
*Main> p
1.0x^0 + 1.0x^1 + -3.0x^2
```

```
*Main> product [x+1, x+2, x+3]
6.0x^0 + 11.0x^1 + 6.0x^2 + 1.0x^3
```

```
*Main> derivada (x*x)
2.0x^1
```

Ejercicios

- Programar `divpoli :: Polinomio -> Polinomio -> Polinomio` que calcula el cociente entre dos polinomios.
- Programar `modpoli :: Polinomio -> Polinomio -> Polinomio` que calcula el resto.
- Programar el algoritmo de Euclides para polinomios (debería poder modificarse el que se hizo hace tiempo).
- Programar `tieneRaicesMultiples :: Polinomio -> Bool`
- Programar `conCoeficientes :: [Float] -> Polinomio`, que dada la lista de los coeficientes de un polinomio devuelva un `Polinomio` con esos coeficientes.